CHAPTER

FIFTEEN

PLITS

The languages we have discussed so far (Distributed Processes and its kin, Ada) use synchronous communication — both the initiator of a communication and the recipient attend to communication. This technique limits the demand for processing resources and simplifies the problem of getting processes to reach synchronous states. Asynchronous communication suggests a greater freedom—the ability to make a request without attending to its completion. Along with this greater freedom comes the risk of unbounded demands for system resources—the possibility that processes will make requests faster than they can be handled.

PLITS (Programming Language In The Sky) is a language based on communication with asynchronous messages. It therefore resembles Actors (Chapter 11). Unlike Actors, PLITS fits the asynchronous mechanisms into an imperative syntax and places a greater emphasis on explicit, interacting processes.

The important primitive objects in PLITS are messages and modules. *Modules* are processes. Modules communicate by sending each other *messages*. PLITS queues and sorts these messages for the receiving process. PLITS also has mechanisms for abstracting and protecting message information, and for selective message reception.

PLITS is the creation of Jerome Feldman and his co-workers at the University of Rochester. Despite its whimsical name, there is an implementation of PLITS. This implementation includes both a high-level language simulator and a distributed system based on message passing and modules.

Messages and Modules

Modules in PLITS communicate asynchronously. PLITS modules enjoy many attributes of true objects: they can be dynamically created and destroyed, and (the names of) modules are themselves a proper data type. PLITS messages are structured association sets. Associated with each module is a queue of messages; modules exercise considerable control over the order in which messages are accepted from the queue.

Communication depends on mutual understanding. To facilitate interprocess communication, PLITS provides a new structured data type, the message. *Messages* are sets of *name-value pairs*. Each such pair is a *slot*. The name field of a slot is an uninterpreted character string and is unique in that message. That is, no name field occurs twice in a given message. The value field is an element of one of a set of primitive unstructured domains, such as integer, real, and module. The value field of a slot cannot be a structured data type, so a message cannot be included in a message. A module declares as *public* the names of the message slots it uses and the types of the slot values. A form of linkage-editing resolves conflicts on slot/types among modules.

PLITS is a foundation on which one can build one's choice of syntax. Following Feldman [Feldman 79], we present a version of "Pascal-PLITS"—PLITS with a Pascal-like syntax. The "." operator of Pascal's record structure extracts message parts, with the slot name serving as the field designator. Name-value pairs (slots) are constructed with operator "∼". Function **message** constructs a message out of slots. Thus, the assignment

```
m := message (day ∼ 135, year ∼ 1983)
```

assigns to variable m a message of two slots. The value of m.day is 135.

PLITS provides many primitives for manipulating messages and modules. The next few paragraphs list these mechanisms. The parenthesized numbers in the text correspond to the lines in Table 15-1, which gives the syntax for each primitive.

A collection of slots can be constructed into a message if their name fields are distinct (1). In PLITS, one can add a slot to a message (changing the value if that slot name is already present) (2), remove a slot from a message (3), change the value of a particular slot (4), and detect the presence (5), or absence (6) of a particular slot in a message. Changing or deleting a nonexistent slot produces an error (3, 4). Modules can manipulate only those message slots to which they have been declared to have public access. Slot names are not a data type so no expression evaluates to a slot name.

Messages can have slots that the recipient of that message cannot access. That is, a module can reference only the slot names that it has declared, but messages may contain other slots. A module can forward an entire message, even if it has access to only a few of its slots. This facility enforces a clever form of se-

Table 15-1 PLITS syntax extensions

1.	**message**(..., $N_i \sim X_i$, ...)	A message constructor function. It returns a message with the given name-value pairs.
2.	**put** $N \sim X$ **in** M	Adds or changes the slot with name N to have value X in message M.
3.	**remove** N **from** M	Deletes the slot with name N from message M. This is an error if M does not have such a slot.
4.	$M.N$:= X	Changes the value of slot with name N to X in message M. This is an error if M does not have such a slot.
5.	**present** N **in** M	True if M has a slot with name N.
6.	**absent** N **in** M	True if M does not have a slot with name N.
7.	**new_transaction**	A function that returns a new transaction key.
8.	**send** M **to** V	Sends the message M to module V.
9.	**send** M **to** V **about** K	Sends M to V. Makes the **about** field of M be the transaction key K.
10.	**receive** M	Removes the next message from the message queue, and assigns it to M.
11.	**receive** M **about** K	Removes the next message from the message queue with a transaction key of K, and assigns it to M.
12.	**receive** M **from** S	Removes the next message from the message queue that was sent from module S, and assigns it to M.
13.	**receive** M **from** S **about** K	Removes the next message from the message queue that was sent from module S with a transaction key K, and assigns it to M.
14.	**pending from** S **about** K	True if there is a message in the queue from S about K. Like **receive**, the **from** and/or **about** clauses are optional.
15.	**new**($ModType$,x_1,...,x_n)	A function that generates a new module of type $ModType$ parameterized by x_1, ..., x_n and returns its name.
16.	**self destruct**	Causes this module to stop processing and "cease to exist."
17.	**extant** V	Does the module V still exist? (Has it already executed **self destruct**?)

curity, where information can be kept from certain modules, but still transmitted by them without resorting to coding tricks or additional communications.

Different models and languages have different methods of organizing and segregating a process's messages. For example, Ada's tasks have multiple entry queues while CSP's "structured data types" require a form of pattern matching for communication. In PLITS, request structuring can be done with transactions. A *transaction* is a unique key. A module can generate a new transaction at will (7); once generated, these transactions are objects of the primitive transaction data type. Every message contains two specific slots: an **about** slot with a transaction key and a **source** slot which specifies the module that sent the message. If a message does not have an explicit **about** slot the system inserts the default transaction key automatically. The system ensures that **source** slots

are correct—that a module cannot "forge" another module's "signature" to a message.

Modules are processes. Each module is the instantiation of a module type. One can have arbitrarily many module instances of that type. Modules have both program and storage. The program portion of a module can execute any of the standard imperative (Pascal-like) control structures. Additionally, modules can compose, decompose, send, and receive messages from other modules. Modules therefore have all the computational power of abstract data types. The data type module is the union of all module types.

The message-sending primitive takes a message and destination and delivers that message to that destination (8). In sending, a process can specify a transaction key (9). **Send** is an asynchronous (send-and-forget) operation; the sending process continues computing after sending. Messages are ultimately routed to the destination module's queue. This requires that there be an "unbounded" queue of unreceived messages kept for each module. Messages from a single sender using a particular transaction key arrive in the order sent and are received in the order sent.* Any module can send a message to any other. Messages are sent by value.

Modules can choose to accept messages in a strictly first-come-first-served order (10). However, there are alternative reception orders. Specifically, a module can specify that a particular reception is to be the next message **about** a particular transaction key (11), the next message **from** a particular source module (12), or the next message **about** a particular key and **from** a particular module (13). This mechanism allows modules some control over the order in which they accept messages, but not as much control as one might imagine. A module might want to receive the message with the highest value on some slot, to exclude messages with a particular transaction key, or, more generally, to accept only messages whose content satisfies some arbitrary predicate. PLITS has a primitive language predicate for determining if there are any pending messages. The pending function can be restricted to messages from a specific source, about a specific transaction key, or both (14).

Programs can dynamically create new instances of a module type (15). Modules can terminate, but only by their own action—by executing the command **self destruct** (16). The **extant** function is true if its argument module has not terminated (17).

Fibonacci numbers Our first example, derived from Feldman [Feldman 79], demonstrates the data structuring, message construction, coroutine, and continuation facilities of PLITS. We imagine three varieties of modules: a type of

* This contrasts with Actors (Chapter 11), where dispatch-order arrival is not guaranteed. Dispatch-order arrival may be difficult to implement when a distributed system allows messages to take different routes to a destination. To ensure this sequencing, modules need to track the history of their communications with other modules.

Fibonacci module that generates Fibonacci numbers, a Printer module that takes a message and prints part of it, and a Seeker module that directs successive Fibonacci numbers from the generator to the Printer (Figure 15-1). At each cycle, the Seeker prompts the Fibonacci module to send the next Fibonacci number to the Printer. The Printer prints the Fibonacci number and sends a synchronization message back to the Seeker. The system repeats this cycle for 100 Fibonacci numbers.

```
program Triangle (Output);
type
  Printer = mod                          -- A Printer is a type of module.
    public                               -- It has access to two message slots.
      continuation : module;             -- The first, "continuation," is the
                                            module destination of the
                                            synchronization pulse.
      object       : integer;            -- The second, "object," is an
                                            integer to be printed.
    var
      m   : message;
      val : integer;
    begin
      while true do                      -- Loop forever, doing:
      begin
        receive m;                       -- Wait for and accept a message.
        val := m.object;                 -- Extract the number.
        writeln (output, val);           -- Print it.
        send m to m.continuation         -- Forward message m to the module
      end                                   in the continuation field of m.
    end;

  Fibonacci = mod                        -- A Fibonacci is a module
    public                               -- that has access to
      whonext : module;                  -- whonext, a continuation, and
      object  : integer;                 -- object, the value to be printed
    var
      this, last, previous : integer;    -- for Fibonacci generation
      m                    : message;
    begin
      last := 0;
      this := 1;
      while true do
      begin
        receive m;                       -- Get the next message. Store it in m.
        previous := last;                -- Compute the next Fibonacci number.
        last     := this;
        this     := last + previous;
```

```
            put object ~ this in m;     -- If there is already a slot of the form
                                           object ~ x in m then replace it;
                                           otherwise, add object ~ this to m.
            send m to m.whonext         -- Send message m to the continuation.
        end
    end;

  Seeker = mod
    const size = 100;
    public
        continuation, whonext: module;
    var
        fibgen   : module;
        printput : module;
        m        : message;
        i        : integer;
    begin
        fibgen   := new(Fibonacci);
        printput := new(Printer);
        put continuation ~ me in m;     -- "me" is the primitive that returns a
                                           module's own name.
        put whonext ~ printput in m;
        for i := 1 to size do           -- Direct the Fibonacci generator's
                                           continuation to do "size" numbers.
        begin
            send m to fibgen;
            receive m    -- If this receive statement is omitted, the numbers
                            are still printed, but the system is no longer
                            synchronized. Greater concurrency results. In this
                            case, the continuation sent by the printer could be
                            omitted.
        end;
    end;

- - - - - - - - - - - - - - - - - - - - main program - - - - - - - - - - - - - - - - - - -
var s: Seeker;
begin
  -- Creating the Seeker module s initiates s. S creates its own Fibonacci and
     Printer modules.
end.
```

Producer-consumer buffer Our second example is a module that acts as a bounded producer-consumer buffer. This module uses transaction keys to control its acceptance of produced values and its presentation of these values to

Figure 15-1 The Fibonacci processes.

consumers. Buffers are created with three parameters: size, the size of the buffer; accept, a transaction key that distinguishes messages that are to be stored in the buffer; and deliver, a transaction key for requests for values. When the buffer receives a request for an element, it sends the first item in the queue to the source of that request.

Modules that use this buffer must not only know the buffer's name, but also have been passed the appropriate transaction key. The process that creates the buffer must create it with two different transaction keys; otherwise, the buffer cannot distinguish producers from consumers.

```
type buffer = mod (size: integer; accept, deliver: transaction)
var
    queue      : array [0 .. size − 1] of message;
    first, last : integer;

procedure intake (w: message);        -- Put this message on the queue.
begin
    last          := (last + 1) mod size;
    queue [last] := w
end;

procedure outplace (w: message);      -- Send the top of the queue to the source
                                         of this message
begin
    first := (first + 1) mod size;
    send queue[first] to w.source
end;

 - - - - - - - - - - - - - - - - - - - - main program - - - - - - - - - - - - - - - - - - -
begin
    first := 0;
    last  := 0;
```

```
    while true do
    begin
        if first = last then                -- queue empty (1)
            receive m about accept          -- Here the buffer is empty. We want
                                               only messages that add to it. We
                                               use the "about" option in receive to
                                               restrict access.
        else
        if (last + 1) mod size = first then     -- queue full (2)
            receive m about deliver             -- The buffer is full. We want
                                                   only requests that consume
                                                   buffer elements.
        else
            receive m;                          -- queue part full (3)
        if m.about = accept
            then intake (m)
            else outplace (m)
    end
end;
```

A program that needs a buffer creates one with **new**, specifying the size of the buffer and transaction keys for accept and deliver. The only action the buffer can take when it is empty is to accept (line 1) and the only action the buffer can take when it is full is to deliver (line 2). When the buffer is partially full, it can both accept and deliver (line 3).

Readers and writers The readers-writers problem requires sharing a resource between two classes of users: readers who can use the resource simultaneously, and writers who require exclusive control. The task is to program a manager that receives requests from readers and writers and schedules their access.

There are two naive ways of approaching the readers-writers problem. The first is to alternate access by a reader and a writer. This solution is unsatisfactory as it excludes concurrent reader access. The alternate extreme is to allow all readers to read and to permit writing only when no reader wants the resource. This scheme has the potential of starving the writers if readers make requests too frequently. (One can give the corresponding priority to writers, threatening the starvation of readers.)

To avoid these pitfalls we take the following approach. We alternate sets of readers and a writer. If both readers and writers are waiting to use the resource, the manager allows all currently waiting readers to read. When they are through, it lets the next writer write. Of course, if only one class of process wants the resource, the manager gives that class immediate service.

Processes must also notify the manager when they are through with the resource. Thus, we imagine that the manager receives three varieties of mes-

Figure 15-2 The states of the readers-writers manager.

sages: read requests, write requests, and release notifications. When a process wants the resource, it sends a message with a slot of the form want ~ PleaseRead or want ~ PleaseWrite to the manager. When it receives a reply with the slot YouHaveIt ~ CanRead or YouHaveIt ~ CanWrite then it has the corresponding access to the resource. When it is through, it sends a message with the slot want ~ ThankYou back to the manager. Thus, we have the enumerated types Request and Permission, declared as

```
type
    Request    = (PleaseRead, PleaseWrite, ThankYou);
    Permission = (CanRead, CanWrite);
```

The manager keeps two queues of requests, one for readers and the other for writers. It alternates between allowing all readers to read and letting the next writer write. We assume that no more than numqueue reader or writer requests are ever pending at any time. The manager keeps its queue using the same queue discipline as the producer-consumer buffer. The manager also keeps a count, using, of modules currently accessing the resource. Figure 15-2 shows the states of the manager.

```
type manager = mod (numqueue: integer);
    queue = array [0 .. numqueue − 1] of module;
public
    want      : Request;
    YouHaveIt : Permission;
var
    using : integer                          -- count of current readers
    m     : message;
```

```
    readqueue, writequeue  : queue;
    readfirst, readlast    : integer;
    writefirst, writelast  : integer;

procedure enqueue (v: module; var q: queue; var last: integer);
begin                                  -- We assume that the queues never overflow.
    last        := (last + 1) mod numqueue;
    queue[last] := v
end;

function dequeue (var q: queue; var first: integer) : module;
begin                          -- We check for an empty queue before calling dequeue.
    first      := (first + 1) mod numqueue;
    dequeue := q[first]
end;

procedure Grant (v: module; p: Permission);
begin
    using := using + 1;
    send message (YouHaveIt ~ p) to v
end;

procedure WaitToClear;     -- This procedure enqueues readers and writers until
begin                         the resource is free.
    while using > 0 do
    begin
        receive m;
        if m.want = ThankYou then
            using := using - 1
        else
            if m.want = PleaseRead then
                enqueue (m.source, readqueue, readlast)
            else           -- must be another write request
                enqueue (m.source, writequeue, writelast)
    end
end;

procedure AcceptReaders;     -- This procedure accepts readers until a writer
begin                           requests the resource.
    repeat
        receive m;
        if m.want = ThankYou then using := using - 1
        else
        if m.want = PleaseRead then Grant (m.source, CanRead)
        else
```

```
        enqueue (m.source, writequeue, writelast);
    until m.want = PleaseWrite
end;

 - - - - - - - - - - - - - - - - - - - - - - main program - - - - - - - - - - - - - - - - - - - -
begin
    readfirst  := 0;    -- Initialize the queue pointers.
    readlast   := 0;
    writefirst := 0;
    writelast  := 0;
    using      := 0;    -- Initially, no process is using the resource.
    while true do       -- The manager loops forever.
    begin               -- The manager is a five-state machine.
            -- If there are no writers waiting, accept all reader requests.
        if writefirst = writelast then AcceptReaders;
            -- Now queue readers and writers until all the readers are done
               with the resource (i.e., using = 0).
        WaitToClear;
            -- Grant access to a writer.
        Grant (dequeue (writequeue, writefirst), CanWrite);
            -- Wait until that writer is done.
        WaitToClear;
            -- Permit all waiting readers to access the resource.
        while not (readfirst = readlast) do
            Grant (dequeue (readqueue, Readfirst), Canread)
            -- And repeat the entire process.
    end
end.
```

Eight queens PLITS supports dynamic creation of new modules. These new modules have no communication restrictions. They can send messages to any other module whose name they come to possess. We use the eight queens problem to illustrate this facility. This problem, investigated by Gauss, requires the placement of eight queens on a chessboard such that no queen can capture any other.* Figure 15-3 shows one solution of the eight queens problem.

One way to solve this problem is to use recursive backtracking. We note that in any solution each row and each column must hold exactly one queen. We place the n^{th} queen on some row of the n^{th} column, check to see if it can be captured by any queen already on the board, and, if it cannot, recursively try to place the remaining queens in the remaining columns. We repeat this process

* In chess, a queen can capture any (opposing) piece that shares the same row, column, or diagonal with it, provided no other piece lies on the path between them.

Figure 15-3 A solution of the eight queens problem.

until the eighth queen is successfully placed. If the queen can be captured, or the recursive attempt fails, we move this queen to another row and repeat the process. If we cannot place the queen on any row, we backtrack, reporting failure to the previous column. We imagine three auxiliary functions, (1) place, which takes a chessboard, row, and column and returns a new board, updated with a queen at that intersection; (2) safe, which is true if its argument board has no mutually attacking queens; and (3) printanswer, which given a board prints the problem solution implied by that board. A pseudo-Pascal function that solves the eight queens problem is as follows:

```
function solve (var brd: chessboard; col: integer) : boolean;
var
    row : integer;
    ans : boolean;
begin
    if safe (brd, col − 1) then
        if col = 9 then
            begin
                printanswer (brd);
                solve := true
            end
        else
            begin
                row := 0;
                repeat
                    row := row + 1;
                    ans := solve (place (brd, row, col), c + 1)
                until (row = 8) or ans;
```

```
                solve := ans
            end
    else
        solve := false
end;
```

Our technique is similar, except that instead of trying each queen placement in turn, we try all the possible queen positions in a column concurrently. (Hence, we do not have to report failures.) The agent of this arrangement is a module, Queen, that receives a message with three slots: (1) a column slot that tells it which column to try to fill; (2) a board slot that contains a representation of those squares of the board already filled; and (3) a continuation slot that contains the identity of the module that eventually prints the answers.

The action of a Queen module is as follows: It receives a message m containing a board and a column. It first checks to see if m.board is safe. If not, the module terminates. If it is, the module determines if it has been asked to fill in the ninth (off-the-edge-of-the-board) column. If so, it has an answer, m.board. It sends m.board to the continuation. It then informs its requestor that it is finished, and terminates.

If this is not a terminal search point, then, for each row, row, the queen module copies its board, adds a new queen at ⟨row, m.column⟩, creates a new queen module, and sends that module the new board, asking it to solve the next column. We assume that a chessboard is a primitive data type that can be included in messages.

After all the generated modules of a queen module have reported completion of the task, the current module reports completion (in the slot done) and terminates. Variable children keeps count of the module's currently computing descendant modules.

```
type queen = mod;
public
    column       : integer;
    board        : chessboard;
    continuation : module;
    done         : boolean;
var
    row          : integer;
    m, problem   : message;
    children     : integer;    -- number of extant modules that this module has
begin                             created
    receive problem;
    if safe (problem.board) then
        if problem.column = 9 then
            send message (ans ~ problem.board) to problem.continuation
```

```
        else
            begin
                children := 8;
                for row := 1 to 8 do
                    begin
                        send message (column ~ problem.column+1,
                                      board ~ place (problem.board,
                                                     row,
                                                     problem.column),
                                      continuation ~ problem.continuation)
                            to new (queen)
                    end;
                while children > 0 do
                    begin
                        receive m;    -- This message is for
                                         synchronization.
                        children := children - 1
                    end
            end;
    send message (done ~ true) to problem.source;
    self destruct
end;
```

A module with an empty board EmptyBoard and the name of a printing continuation Printing could have the 92 solutions to the eight queens problem sent to Printing (and a termination confirmation message sent to itself) with the command

```
send message (column ~ 1,
              board ~ EmptyBoard,
              continuation ~ Printing)
    to new (queen);
```

Perspective

PLITS is based on processes and messages. Processes are objects; they possess program and storage and can be dynamically created and destroyed. Process names can be passed between processes.

Processes communicate by asynchronously sending each other messages. These messages are transmitted by call-by-value (copying). The underlying system keeps an unbounded queue of unreceived messages for each process. The process can check the size of its queue or treat it as several subqueues, simultaneously organized by sender and subject.

Turning constant into variable is a boon to most programming activities (though one sometimes trades efficiency for this flexibility). PLITS's treatment of processes as a data type, to be created, referenced, and destroyed is an example of such a generalization. PLITS correctly recognizes that an asynchronous distributed system must treat process names as a proper data type.

Feldman and his co-workers at Rochester are in the process of implementing a distributed system founded on the ideas of PLITS. While their short term goals have been directed at organizing a varied collection of computers, they clearly share many of the long-term goals of coordinated computing.

PROBLEMS

15-1 If PLITS had not provided the primitives **self destruct** and **extant**, how could a programmer achieve the same effect?

15-2 How many elements can fit into the bounded buffer before it is full?

15-3 The program for the readers-writers problem assumed that no more than a constant number (numqueue) of reader or writer requests would ever be pending. Modify that program to remove this restriction. (Hint: Use transaction keys.)

15-4 Similarly, the readers-writers program used two internal, finite queues. Modify that program to use the message queue instead.

15-5 Generalize the eight queens problem to the n-queens problem over an $n \times n$ chessboard.

15-6 The sorcerer's apprentice: The eight queens program generates all solutions to the problem. Modify the program to stop (reasonably soon) after the first solution has been found. (*Muchnick*)

REFERENCES

[**Feldman 79**] Feldman, J. A., "High Level Programming for Distributed Computing," *CACM*, vol. 22, no. 6 (June 1979), pp. 353–368. This paper describes PLITS. It also discusses implementation issues for PLITS-like systems and several pragmatic issues (such as typing, assertions, and verification) not specific to PLITS, but of concern to the general problem of programming.

[**Wirth 76**] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey (1976). This book is a good introduction to programming style and data structures. On pages 143–147, Wirth presents an excellent description of the eight queens problem and a Pascal program that solves it.